

DESIGN AND IMPLEMENTATION OF A FULLY DISTRIBUTED CACHING ALGORITHM ON AN NDN SYSTEM

An Undergraduate Research Scholars Thesis

by

ABIGAIL DOWD

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor:

Dr. I-Hong Hou

May 2018

Major: Electrical Engineering

TABLE OF CONTENTS

	Page
ABSTRACT.....	1
NOMENCLATURE	2
CHAPTER	
I. INTRODUCTION	3
Named Data Networking	3
Benefits of NDN	4
Previous Work	4
II. METHODS	7
Model	7
Caching Algorithm.....	7
Experimental Procedures	8
NDN Testbed.....	13
III. RESULTS	15
Cost Evaluation	15
Numerical Results	15
IV. CONCLUSION.....	18
REFERENCES	21

ABSTRACT

Design and Implementation of a Fully Distributed Caching Algorithm on an NDN System

Abigail Dowd

Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. I-Hong Hou

Department of Electrical and Computer Engineering
Texas A&M University

ICN (Information Centric Networking) is a new method of storing and accessing data on the internet which focuses on the content itself rather than the IP (Internet Protocol) address where the content is stored. ICN enables both in-network caching and name-based data retrieval [10]. This allows for better usage of edge cloud resources, giving the user a faster response time as some data requests and services may be handled locally [2]. NDN (Named Data Networking) is a specific type of ICN which locates and delivers content based on the associated data name rather than using the source or destination host addresses [15]. For NDN to be most beneficial, we need to implement efficient caching algorithms that consider the needs of many users in a network. To address this need, we have developed a caching algorithm for an NDN network in a tree topology. It is fully distributed and makes storage and eviction decisions at each router based on the number of hops needed to retrieve the data and the popularity of the data at that router. The total number of hops taken by all data during the testing period determined the algorithm's true cost. We tested our algorithm using an NDN testbed and compared its true cost with another commonly used algorithm, LRU (Least Recently Used), under the same conditions. Our cost-based policy incurred a lower true cost in all test cases, with average savings ranging from 9% to 19% depending on cache size and popularity distribution. The cost policy performed particularly well in comparison to LRU when the cache size was small.

NOMENCLATURE

ICN	Information Centric Networking
IP	Internet Protocol
LRU	Least Recently Used
FIFO	First In First Out
NDN	Named Data Networking
NFD	NDN Forwarding Daemon
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

CHAPTER I

INTRODUCTION

Named Data Networking

The current model that the internet is built upon is an IP (Internet Protocol) system in which data packets are associated with specific IP addresses and are transferred via direct host-to-host communication. While this system—storing data at a specific location and then retrieving the data from that location upon request—worked in the past, the volume and complexity of the content being created and redistributed today makes the IP-based system extremely inefficient [1]. NDN (Named Data Networking) offers a new method of publishing and retrieving content that works more effectively with the way most consumers use the internet today. Rather than assigning content to an IP address, each piece of data is named and can later be requested by its name. This reduces the delivery of large amounts of redundant data by allowing for interest checking and in-network caching which more efficiently utilizes edge cloud resources [11]. In other words, data can be stored within the network and may be retrieved quickly without travelling all the way to the source.

NDN was created as a method which may replace IP on new networks or run on top of IP on existing systems for which it is not practical or desirable to change the architecture. Because the internet as it exists today is entirely dependent on IP, the most commonly used and researched implementation of NDN is as a wrapper for IP. This means that an NFD (NDN Forwarding Daemon) must set up the links needed for NDN transmission by utilizing either UDP (User Datagram Protocol) or TCP (Transmission Control Protocol) sockets. However, these sockets are hidden from NDN, so that it is ignorant of the existence of IP addresses and may

operate only based on data names, as was intended. The goal of this project is to utilize NDN on a WiFi network to test a new in-network caching algorithm, which operates in a fully distributed manner and considers the needs of all on-path routers.

Benefits of NDN

Name-based routing, packet-level caching, and cache look-up are a few methods currently employed by NDN routers [10]. However, all of these cause huge overhead on the routers. Since in-network caching is one of the major benefits of adopting NDN, efficient caching which works well with the NDN architecture is essential. To achieve this, we plan to configure an NDN router on which to test and modify caching algorithms which minimize overhead and use edge cloud resources more efficiently than the current NDN caching algorithms.

Previous Work

Traditionally caching algorithms were used to determine which data should be stored in the cache of a single machine. In that case, only the needs of that machine were considered and algorithms like LRU(Least Recently Used) and FIFO(First In First Out) were developed. Conversely, in-network caching algorithms must consider the needs of many users and make decisions that benefit the entire network. However, LRU is still heavily used due to its simplicity, low overhead, and decent return. This means any algorithm which is more complex needs to show major benefits in terms of speed of service, redundancy reduction, and reduced stress on the network.

In an attempt to show the benefits of more complex designs, two main classes of algorithms are being explored: on-path and off-path caching algorithms [6]. On-path caching algorithms typically seek to store data along the path between the source and destination. The

advantage of this method is that it minimizes information flow between nodes thereby reducing stress on the network. The disadvantage is that it may not utilize cache space along less frequently used paths. Most on-path caching algorithms attempt to assign or calculate some probability p for each node, which is used to determine which data should be cached and which should be evicted [8]. In some attempts, p is a fixed value [3,5,9]. Other on-path caching algorithms assign cache weights which are factored into the caching decisions [13]. Finally, many papers take a graphical approach and consider various measures of centrality when calculating the probability to cache for each node [14].

Off-path caching algorithms usually focus on maximizing the hit rate of all caches in the network by storing data in caches that do not lie directly between the source and destination. Although these algorithms do improve cache hit rates, they also require more complex methods of locating off-path data as described in [4]. Most collaborative caching methods use a *Name Resolution Server* (NRS) or a similar structure to keep track of what data is stored in which caches [4]. This global approach can be very expensive since the NRS must be updated frequently in order to maintain accurate information. Other models based on total traffic cost [16] or degree centrality [17] were also proposed. Again, these algorithms typically have higher cache hit rates than either traditional or on-path caching algorithms, but also have high overheads and may require additional packets to be sent between nodes to share information about their cache contents [12].

The algorithm we will present is an on-path caching algorithm that considers the distance to retrieve a piece of data as well as the popularity at all of the routers which could potentially request the data. In other words, the algorithm attempts to consider every user who could benefit or suffer from the caching decision.

Updating the popularity and distance to data in a fully distributed network can be difficult since each node only has information it receives through interest and data packets. We have addressed the challenges of making informed decisions in a system with limited information sharing capabilities by creating a semi-collaborative algorithm with a known topology. In cases where the network topology is completely unknown, an algorithm such as the one proposed in [7] for an arbitrary topology must be used. However, in the cases where we know the topology, our algorithm can make more effective decisions with less information transmitted along with the interest and data packets.

In this paper, we will begin by discussing the system model. We will show the calculations of popularity, distance to data, and retrieval cost and describe their role in the storage or eviction decisions. We will explain how the algorithm was implemented in the NDN code. Finally, we will show that the cost policy developed in this paper incurred a lower cost than LRU in all test cases. A caching algorithm that reduces the total number of transmissions in a network while still fulfilling all requests will simultaneously improve users' experiences by delivering popular data more quickly and reduce total traffic on the network.

CHAPTER II

METHODS

Model

We considered a network with a total of i nodes, each of which has a cache with a constant capacity. We denote a single node by n_i and the set of nodes by N . The nodes were arranged in a tree topology and allowed for only unidirectional data transmission from the source to edges of the tree and vice versa. Data was requested and interest packets generated at each node at a constant rate. We denote each piece of data by d_j and the set of all data by D . A constant popularity for each piece of data was assigned using the Zipf distribution. The requests for each piece of data were generated with a probability proportional to the data's popularity. When an interest packet was generated, a node first checked its own cache, and then forwarded the interest if it did not have the requested piece of data. The interest was then forwarded up the tree until the data was located. The data was sent back down the tree and had the opportunity to be stored at nodes on the path.

Caching Algorithm

We implemented a combined storage and eviction cache policy, which stores content with the highest retrieval cost. C_{ij} is the retrieval cost for data j at node i . It is determined based on the popularity of the data and the number of hops needed to retrieve the data if it is not stored locally (Equation 1). P_{ij} is the total popularity of data j at and below node i . L_{ij} is the distance, measured in hops, between data j and node i (Figure 1).

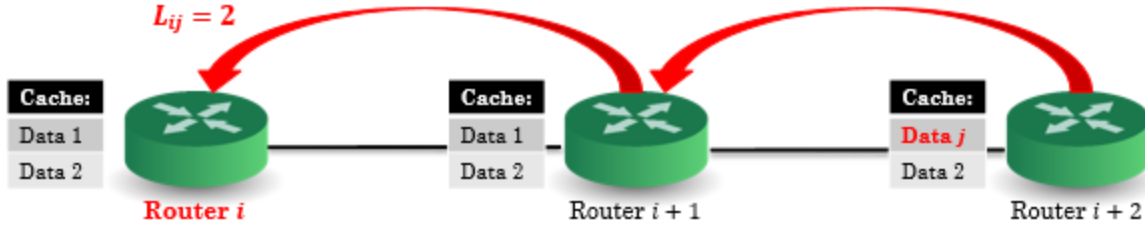


Figure 1. Description of L_{ij}

$$C_{ij} = L_{ij}P_{ij}$$

$$= L_{ij} \sum_{n: n \leq i} p_{nj}$$

Equation 1. Retrieval cost for data j at node i .

For a new data d_k arriving at a node i , C_{ik} will be calculated using Equation 1. If C_{ik} is greater than C_{in} for any data d_n already cached at node i , then data d_k will be stored while data d_n is evicted.

Experimental Procedures

We began by implementing an NDN edge cloud system on several Ubuntu machines which were used for testing caching algorithms. This was accomplished by utilizing existing code to run an NDN Forwarding Daemon (NFD) which communicates between the existing IP system and the NDN code that runs on top of it. After installing NFD and establishing basic communication between machines, we isolated the code that controls the caching protocol and tested some commonly used caching algorithms prior to implementing our own on the NDN architecture.

Popularity and Distance to Data

In order to facilitate the calculation of retrieval cost as described in Equation 1, we kept track of the local popularity P_{nj} for each piece of data. For data currently stored in the cache, we

also needed to know distance to data L_{ij} and the summed popularity P_{ij} (Equation 1). We tracked the local popularity by introducing a metacache, which is simply a cache that only stores data names. The purpose of this metacache was to track the relative popularity of data at a particular node without actually storing the data. The metacache reacted to interest packets received and updated the position of the requested data using the climb algorithm, which moves an entry up one position each time a request for it is received. All entries at or below that position are shifted down as demonstrated in Figure 2.

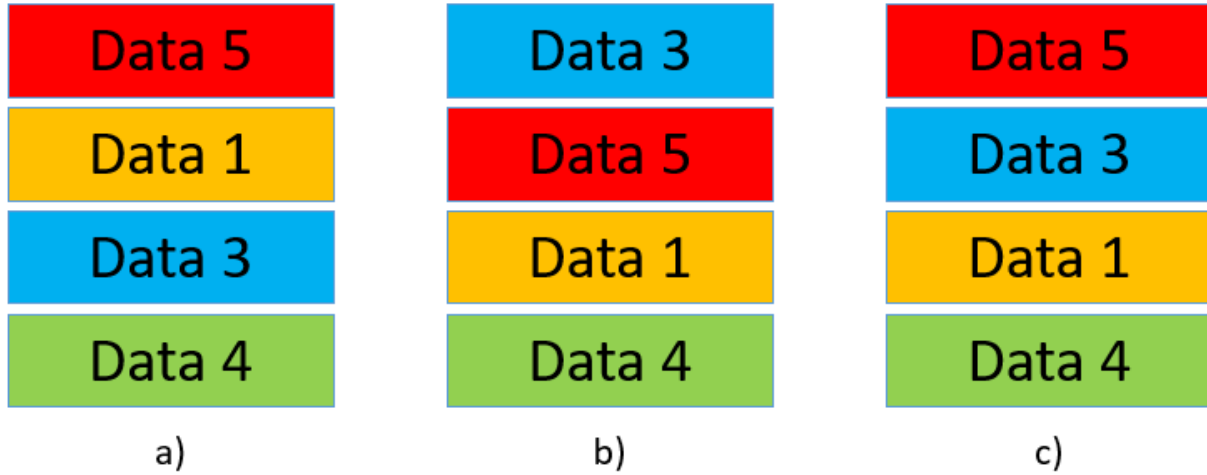


Figure 2. Each data name's position in the metacache is updated when an interest packet is received. a) Initial state of the metacache. b) Two interests for Data 3 are received. c) One interest for Data 5 is received.

The position of a data name in the metacache is directly related to the relative popularity of that piece of data at that node. The Zipf distribution is used to estimate the frequency of events and has been shown to be a good model for data request rates on the internet. It works by assuming that the frequency of each event is a function of the frequency of the most popular event. In this case, we will assume that the most popular data is in the first position in the metacache. The local popularity for each piece of data is estimated by mapping the position of

the data name in the metacache to a popularity value using the Zipf distribution as shown in Equation 2 and Figure 3.

$$p = \frac{1}{(m_{ij} + 1)^\alpha}$$

Equation 2. Zipf equation for popularity p , given m_{ij} the position of data j at router i in the distribution and the Zipf parameter α .

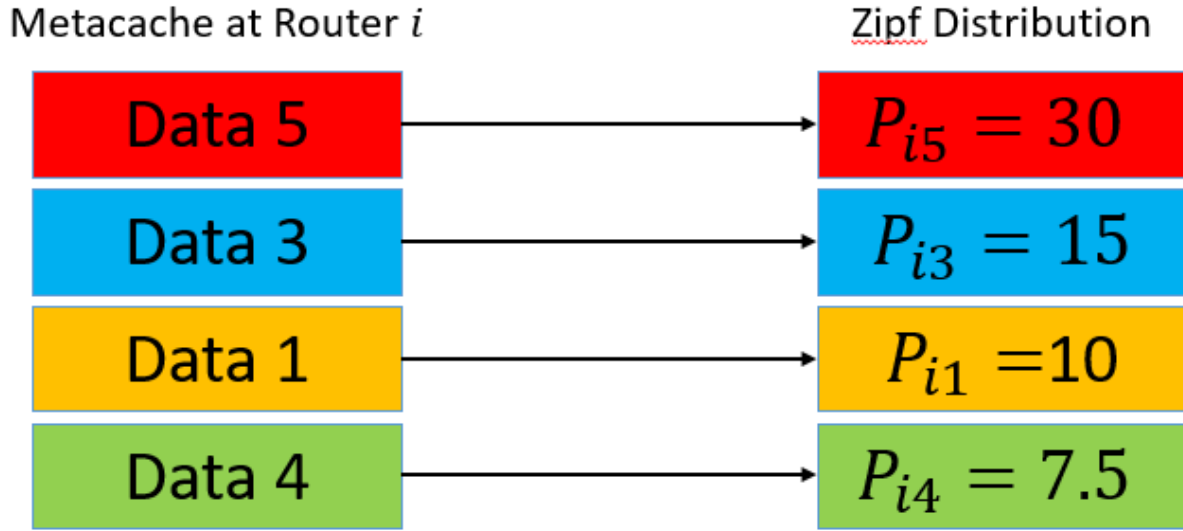


Figure 3. If router i receives 60 interests/minute and $\alpha = 1$, then P_{ij} will be estimated for each piece of data as shown.

When an interest packet was created, the local popularity of the requested data was attached as a tag. For each router the interest packet passed through, the local popularity was added to the popularity tag. When the requested data was located, this value— P_{ij} as described in Equation 1—was transferred from the interest packet to the data packet. As the data packet traveled back to the original requester, the popularity tag was extracted, the retrieval cost was calculated, and the caching decision was made at each node on the return path. To appropriately

scale the popularity tag for returning data packets, the local popularity at each node was subtracted before the data was forwarded.

In addition to the local popularity, a table was created on each node to keep track of total popularity P_{ij} and distance to data L_{ij} for each piece of data currently stored in the cache. These two values were updated using the popularity and distance to data tags on incoming interest and data packets. Interest packets for data already stored locally were forwarded with probability 1% to update hop count periodically. A summary of the notation described in this section is given in Table 1.

Table 1. Notation Summary

C	Total cache space for router i
k	Cache capacity
M	Total metacache space for router i
m_{ij}	Position in metacache i of data j
N	Full set of nodes in the system
n_i	Node i
D	Full set of data in the system
d_j	Data j
p_{ij}	Local popularity at router i for data j
P_{ij}	Total popularity at router i for data j
L_{ij}	Length of path (in hops) for router i to retrieve data j
C_{ij}	Retrieval cost at router i for data j
T	True cost; total number of hops taken by all data in system
H_j	Total number of hops taken by data j

Algorithm Implementation

The cost calculation and cache policy described above were implemented in NDN using the logic summarized below. Algorithm 1 describes how incoming interest and data packets are handled at each node. Algorithm 2 describes the actual storage and eviction policy based on retrieval cost, C_{ij} .

Algorithm 1 Protocol for Received Packet

```
1: Execute the following at each  $n \in N$ 
2: for packet received do:
3:   If packet is interest then:
4:     Update metacache position  $m_{ij}$ 
5:     Update popularity tag  $P_{ij}$  by adding local popularity  $P_{nj}$ 
6:     If  $d_j \in C$  then:
7:       Update  $P_{ij}$  in table
8:       Serve request
9:     else:
10:      Forward interest
11:    end If
12:  end If
13:  If packet is data then:
14:    If  $d_j \in C$  then:
15:      Update  $L_{ij}$  in table
16:    else:
17:      Calculate  $C_{ij}$  (Equation 1)
18:      Execute Algorithm 2
19:    end If
20:    If  $d_j \in C$  then:
21:      Set distance to data tag to 1
22:    end If
23:    Forward data
24:  end If
25: end for
```

Algorithm 2 Simple Cache Storage and Eviction

```
1: Input: Retrieval cost of new data  $C_{ij}$ 
2: Output: Cache received data  $d_j$ 
3: If  $\min_{k: d_k \in C} C_{ik} < C_{ij}$  then:
4:   Evict  $d_{\arg \min_{k: d_k \in C} C_{ik}}$ 
5:   Cache  $d_j$ 
6: end If
```

NDN Testbed

A small NDN network was constructed using Ubuntu machines connected via WiFi. The machines themselves acted as routers and were arranged in a tree topology as shown in Figure 4.

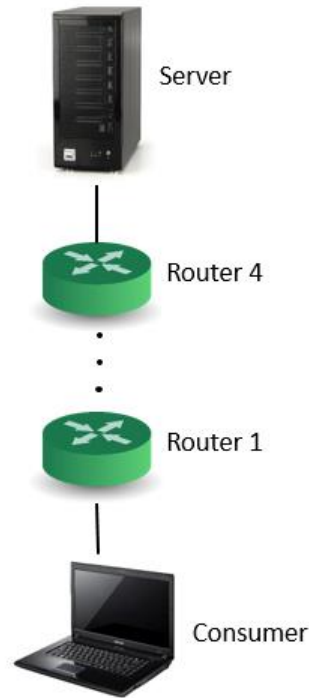


Figure 4. Testbed topology.

Four routers were connected in a row and a single consumer was placed at the end. The consumer generated interest packets using a Zipf distribution with 100 possible pieces of data and a Zipf parameter of 0.7, 0.8, 0.9 or 1.0. The cost policy described above made storage and eviction decisions with the assumption that the popularity distribution of the data resembled a Zipf distribution with a parameter equal to 0.8 for all cases. A different Zipf parameter was used to generate interests for each test case to show that the policy would still perform well when the true popularity distribution varied. A variety of caches sizes were also tested to show that the cost policy performs well regardless of the amount of memory available. Table 2 gives a complete description of all test parameters.

Table 2. Test parameters.

Topology:	Tree with single consumer at leaf
Distribution:	$Zipf(\alpha, N)$, $\alpha = 0.7, 0.8, 0.9, 1.0$, $N = 100$ for all cases
Rate of requests:	30 interests/min
Total requests during testing period:	500 interests
Cache size:	$k = 3, 5, 7, 10$
Metacache size:	$3 \times k$

CHAPTER III

RESULTS

Cost Evaluation

During testing, true cost (T) was calculated using the sum of hop counts required by all data packets in the system during the testing period (Equation 3).

$$T = \sum_{j \in J} H_j$$

Equation 3. True cost, where J is the complete set of data packets transmitted during the testing period and H_j is the total number of hops taken by a data $j \in J$.

For every data packet delivered to a node during the testing period the hop count or the total distance traveled by that packet was recorded. These values were summed across all data packets at all nodes producing the true cost of the algorithm. In this experiment, we chose to measure hop count instead of cache hit rate because our goal was to reduce the total number of transmissions thereby reducing stress on the network. However, the true cost and the cache hit rate are related since every cache miss would necessitate an additional hop to recover the requested data.

Numerical Results

Our cost-based policy's performance was compared to LRU, which is frequently used in networking applications due to its adaptability, low overhead, and reasonably good performance. Both policies were tested using various Zipf parameters and cache sizes. In each case, the true cost T was measured, and the results are displayed below (Figure 5).

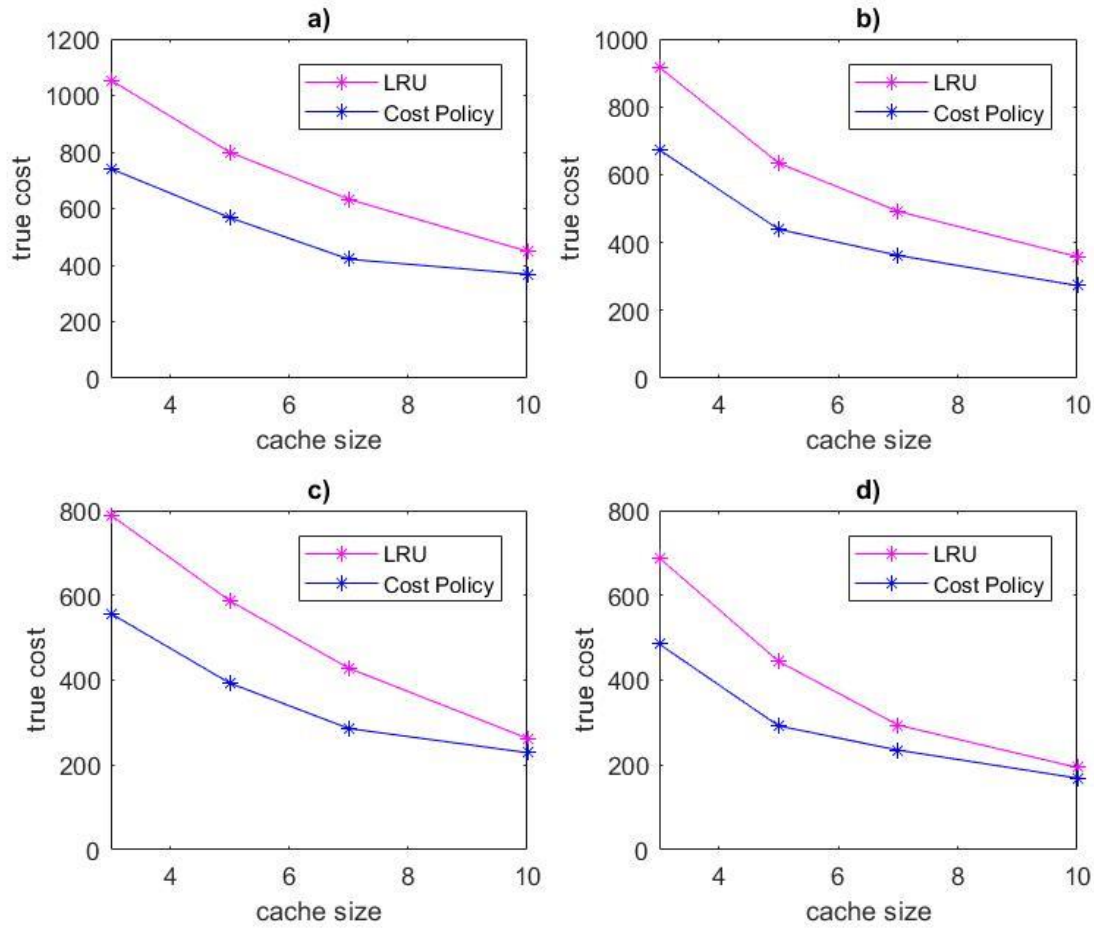


Figure 5. Total cost vs. cache size. a) $Zipf(\alpha = 0.7, N = 100)$, b) $Zipf(\alpha = 0.8, N = 100)$, c) $Zipf(\alpha = 0.9, N = 100)$, d) $Zipf(\alpha = 1.0, N = 100)$.

Table 3. Average percent difference in true cost for each cache size.

Cache Size	Percent Difference in True Cost T
3	16.83 %
5	18.89 %
7	16.59 %
10	9.30 %

For the cases tested, the cost policy and LRU performed most similarly when the Zipf parameter of the generated data distribution and the cache size were large. With a Zipf parameter of 1.0 and a cache size of 10, the cost policy's true cost was 6.98% lower than that of LRU. This represents the worst case scenario of all cases tested. However, on average the cost policy's true cost was between 9 and 19% lower than that of LRU, and the cost policy outperformed LRU in all cases (Table 3).

CHAPTER IV

CONCLUSION

As the amount of content on the internet grows and the rate at which that content is requested increases, more efficient methods of handling data on the internet will become essential. NDN is a potential solution as it allows for name-based retrieval and in-network caching. These attributes help utilize edge cloud resources more effectively and make popular content more accessible to the users who request it. The goal of this project was to design and implement an in-network caching algorithm which reduces the average cost of fulfilling content requests.

Algorithms currently being used for in-network caching include traditional caching algorithms like LRU, on-path caching algorithms, and off-path caching algorithms. In this paper, we introduced a semi-collaborative on-path caching algorithm which considers each piece of data's potential retrieval cost for storage and eviction decisions. The cost policy approach has several advantages over the other algorithms currently being used. All information needed to make the cache storage and eviction decisions is transmitted via tags on the interest and data packets. This allows for collaboration between nodes without transmission of additional data packets. The cost policy considers not only the popularity of each piece of data at the node making the caching decision, but also the popularity at all nodes which may be served by that cache. Finally, the cost policy considers the distance to retrieve a particular piece of data if that data were not stored locally. The total popularity and the distance were used to calculate a retrieval cost for each piece of data, which determined whether or not it was stored. During testing, the true cost of each algorithm was calculated by summing the total number of hops

taken by each piece of data in the system. A more effective caching policy should have a lower true cost as the most popular data would be stored closest to the users requesting it. As expected, the cost policy generated a lower true cost than LRU in all test cases.

The two parameters varied during testing were the parameter of the Zipf distribution used to generate requests for data and the cache size in each router. Both the cost policy and LRU performed better when the requested data was generated from a Zipf distribution with a higher parameter. This is simply because the popularity of the data was distributed less evenly. In other words, unpopular data is requested less frequently leading to fewer cache misses overall. However, the cost policy performed well relative to LRU for each Zipf parameter tested. As cache size decreased, true cost increased for both algorithms. However, the cost policy performed particularly well compared to LRU for medium to small cache sizes. There appeared to be an optimal cache size for the cost policy around 5% of the total amount of data in the system. As the cache size increased from 5%, the percent difference between the two policies shrank. We expect that if we continued to increase cache size, the cost policy and LRU would have similar true costs. This implies that at worst the cost policy incurs the same cost as LRU, and at best reduces the cost by about 19%.

In the future, this policy could be improved further by introducing some randomness into the caching decision to prevent the cache contents from converging to a local minimum rather than the optimal solution. Instead of always storing the data with the maximum retrieval cost, the retrieval cost would indicate a probability of storage. This would provide an opportunity to continue to modify the cache contents even when the algorithm has found a minimum cost solution. In addition to introducing randomness into the system, the policy could also be adapted for other network topologies.

In this paper, we have shown that a cache policy which considers the popularity of a piece of data at each router in a system as well as the cost of retrieving that data upon request performs better than traditional caching algorithms which only consider the frequency of requests at a single router. The cost policy reduced the average number of hops taken by interest and data packets alleviating the stress on the network caused by unnecessary transmissions. Utilizing efficient algorithms like the cost policy presented here will become increasingly important as in-network caching becomes more widespread and the number of requests for content on the internet continue to increase.

REFERENCES

- [1] “A Conversation with Van Jacobson,” *A Conversation with Van Jacobson - ACM Queue*, 08-Jan-2009. [Online]. Available: <http://queue.acm.org/detail.cfm?id=1508215>. [Accessed: 29-Aug-2017].
- [2] M. Amadeo, C. Campolo, and A. Molinaro, “NDNe: Enhancing Named Data Networking to Support Cloudification at the Edge,” *IEEE Communications Letters*, vol. 20, no. 11, pp. 2264–2267, 2016.
- [3] S. Arianfar, P. Nikander, and J. Ott, “On content-centric router design and implications,” *Proceedings of the Re-Architecting the Internet Workshop on - ReARCH 10*, 2010.
- [4] S. Bayhan, L. Wang, J. Ott, J. Kangasharju, A. Sathiaselalan, and J. Crowcroft, “On Content Indexing for Off-Path Caching in Information-Centric Networks,” *ACM-ICN '16*, Sep. 2016.
- [5] W. Chai, D. He, I. Psaras, and G. Pavlou, “Cache ”less for more” in information-centric networks,” *Proceedings of the 11th International IFIP TC 6 Conference on Networking*, Brooklyn, NY, USA, May 2012.
- [6] M. Draxler and H. Karl, “Efficiency of On-Path and Off-Path Caching Strategies in Information Centric Networks,” *2012 IEEE International Conference on Green Computing and Communications*, 2012.
- [7] S. Ioannidis and E. Yeh, “Adaptive Caching Networks with Optimality Guarantees,” *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science - SIGMETRICS 16*, 2016.
- [8] A. Ioannou and S. Weber, “Towards on-path caching alternatives in Information-Centric Networks,” *39th Annual IEEE Conference on Local Computer Networks*, 2014.
- [9] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard, “Networking named content,” *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*, Rome, Italy, December 2009, pp. 1–12.

- [10] D. Kim and Y. Kim, “Enhancing NDN feasibility via dedicated routing and caching,” *Computer Networks*, vol. 126, pp. 218–228, Jul. 2017.
- [11] Y. Kim, Y. Kim, J. Bi, and I. Yeom, “Differentiated forwarding and caching in named-data networking,” *Journal of Network and Computer Applications*, pp. 155–169, Dec. 2015.
- [12] T. Mick, R. Tourani, and S. Misra, “MuNCC.” *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking - ACM-ICN 16*, 2016.
- [13] I. Psaras, W. Chai, and G. Pavlou, “Probabilistic in-network caching for information-centric networks,” *Proceedings of the 2nd Workshop on Information-Centric Networking*, Orlando, FL, USA, March 2012.
- [14] G. Rossini and D. Rossi, “On sizing ccn content stores by exploiting topological information,” *Proceedings of the 1st Workshop on Emerging Design Choices in Name Oriented Networking (NOMEN '12)*, pp. 280–285, March 2012.
- [15] K. Shilton, J. A. Burke, K. Claffy, and L. Zhang, “Anticipating policy and social implications of named data networking,” *Communications of the ACM*, vol. 59, no. 12, pp. 92–101, Jan. 2016.
- [16] V. Sourlas, L. Gkatzikis, P. Flegkas, and L. Tassiulas. “Distributed cache management in information-centric Networks,” *IEEE Trans. on Network and Service Management*, pp. 286–299, 2013.
- [17] J. Wang, J. Zhang, and B. Bensaou. “Intra-AS cooperative caching for content-centric networks” *In ACM ICN*, pp.61–66, 2013.